

Lecture 15

# CIS 341: COMPILERS

# Announcements

- HW4: OAT v. 1.0
  - Parsing & basic code generation
  - **Due: March 28<sup>th</sup>**
  - **START EARLY!**
- Midterm Exam
  - Grading almost finished. We expect to release the results on gradescope by Thursday

**Note new  
Due Date!**

# Inference Rules

- We can read a judgment  $G;L \vdash e : t$  as “the expression  $e$  is well typed and has type  $t$ ”
- For any environment  $G$ , expression  $e$ , and statements  $s_1, s_2$ .

$$G;L;rt \vdash \text{if } (e) s_1 \text{ else } s_2$$

holds if  $G;L \vdash e : \text{bool}$  and  $G;L;rt \vdash s_1$  and  $G;L;rt \vdash s_2$  all hold.

- More succinctly: we summarize these constraints as an *inference rule*:

Premises	$G;L \vdash e : \text{bool}$	$G;L;rt \vdash s_1$	$G;L;rt \vdash s_2$
Conclusion	$G;L;rt \vdash \text{if } (e) s_1 \text{ else } s_2$		

- This rule can be used for *any* substitution of the syntactic metavariables  $G$ ,  $e$ ,  $s_1$  and  $s_2$ .

# Compilation As Translating Judgments

- Consider the source typing judgment for source expressions:

$$C \vdash e : t$$

- How do we interpret this information in the target language?

$$\llbracket C \vdash e : t \rrbracket = ?$$

- $\llbracket t \rrbracket$  is a target type
- $\llbracket e \rrbracket$  translates to a (potentially empty) sequence of instructions, that, when run, computes the result into some operand
- INVARIANT: if  $\llbracket C \vdash e : t \rrbracket = \text{ty}, \text{operand}, \text{stream}$   
then the type (at the target level) of the operand is  $\text{ty} = \llbracket t \rrbracket$

# Example

- $C \vdash 341 + 5 : \text{int}$       what is  $\llbracket C \vdash 341 + 5 : \text{int} \rrbracket$  ?

$\llbracket \vdash 341 : \text{int} \rrbracket = (\text{i64}, \text{Const } 341, [])$

$\llbracket \vdash 5 : \text{int} \rrbracket = (\text{i64}, \text{Const } 5, [])$

-----  
 $\llbracket C \vdash 341 : \text{int} \rrbracket = (\text{i64}, \text{Const } 341, [])$

-----  
 $\llbracket C \vdash 5 : \text{int} \rrbracket = (\text{i64}, \text{Const } 5, [])$

-----  
 $\llbracket C \vdash 341 + 5 : \text{int} \rrbracket = (\text{i64}, \%tmp, [\%tmp = \text{add i64 (Const 341) (Const 5)}])$

# What about the Context?

- What is  $\llbracket C \rrbracket$ ?
- Source level  $C$  has bindings like:  $x:\text{int}, y:\text{bool}$ 
  - We think of it as a finite map from identifiers to types
- What is the interpretation of  $C$  at the target level?
- $\llbracket C \rrbracket$  maps source identifiers, “ $x$ ” to source types and  $\llbracket x \rrbracket$
- What is the interpretation of a variable  $\llbracket x \rrbracket$  at the target level?
  - How are the variables used in the type system?

$$\frac{x:t \in L}{G;L \vdash x : t} \quad \text{TYP\_VAR}$$

as expressions  
(which denote values)

$$\frac{x:t \in L \quad G;L \vdash \text{exp} : t}{G;L;rt \vdash x = \text{exp}; \Rightarrow L} \quad \text{TYP\_ASSN}$$

as addresses  
(which can be assigned)

# Interpretation of Contexts

- $\llbracket C \rrbracket$  = a map from source identifiers to types and target identifiers
- INVARIANT:  
     $x:t \in C$       means that
  - (1)     $\text{lookup } \llbracket C \rrbracket x = (t, \%id\_x)$
  - (2)    the (target) type of  $\%id\_x$  is  $\llbracket t \rrbracket^*$     (a pointer to  $\llbracket t \rrbracket$ )

# Interpretation of Variables

- Establish invariant for expressions:

$$\left[ \frac{x:t \in L}{G;L \vdash x : t} \text{ TYP\_VAR} \right] = (\%tmp, [\%tmp = \text{load } i64* \%id\_x])$$

as expressions  
(which denote values)

where  $(i64, \%id\_x) = \text{lookup } \llbracket L \rrbracket x$

- What about statements?

$$\left[ \frac{x:t \in L \quad G;L \vdash exp : t}{G;L;rt \vdash x = exp; \Rightarrow L} \text{ TYP\_ASSN} \right] = \text{stream @}$$

as addresses  
(which can be assigned)

$[\text{store } \llbracket t \rrbracket \text{ opn}, \llbracket t \rrbracket * \%id\_x]$

where  $(t, \%id\_x) = \text{lookup } \llbracket L \rrbracket x$   
and  $\llbracket G;L \vdash exp : t \rrbracket = (\llbracket t \rrbracket, \text{opn}, \text{stream})$



## Other Judgments?

- Statement:  
 $\llbracket C; rt \vdash \text{stmt} \Rightarrow C' \rrbracket = \llbracket C' \rrbracket, \text{stream}$
- Declaration:  
 $\llbracket G; L \vdash \text{var } x = \text{exp} \Rightarrow G; L, x:t \rrbracket = \llbracket G; L, x:t \rrbracket, \text{stream}$

INVARIANT: stream is of the form:

```
stream' @  
[ %id_x = alloca  $\llbracket t \rrbracket$ ;  
  store  $\llbracket t \rrbracket$  opn,  $\llbracket t \rrbracket^* \text{\%id\_x}$  ]
```

when  $\llbracket G; L \vdash \text{exp} : t \rrbracket = (\llbracket t \rrbracket, \text{opn}, \text{stream}')$

- Rest follow similarly



# COMPILING CONTROL

# Translating while

- Consider translating “while(e) s”:
  - Test the conditional, if true jump to the body, else jump to the label after the body.

$\llbracket C; \text{rt} \vdash \text{while}(e) \ s \Rightarrow C' \rrbracket = \llbracket C' \rrbracket,$

```
lpre:
    opn =  $\llbracket C \vdash e : \text{bool} \rrbracket$ 
    %test = icmp eq i1 opn, 0
    br %test, label %lpost, label %lbody
lbody:
     $\llbracket C; \text{rt} \vdash s \Rightarrow C' \rrbracket$ 
    br %lpre
lpost:
```

- Note: writing `opn =  $\llbracket C \vdash e : \text{bool} \rrbracket$`  is pun
  - translating  $\llbracket C \vdash e : \text{bool} \rrbracket$  generates *code* that puts the result into `opn`
  - In this notation there is implicit collection of the code

# Translating if-then-else

- Similar to while except that code is slightly more complicated because if-then-else must reach a merge and the else branch is optional.

$$\llbracket C; \text{rt} \vdash \text{if } (e_1) \ s_1 \ \text{else} \ s_2 \Rightarrow C' \rrbracket = \llbracket C' \rrbracket$$

```
    opn =  $\llbracket C \vdash e : \text{bool} \rrbracket$ 
    %test = icmp eq i1 opn, 0
    br %test, label %else, label %then
then:
     $\llbracket C; \text{rt} \vdash s_1 \Rightarrow C' \rrbracket$ 
    br %merge
else:
     $\llbracket C; \text{rt} \vdash s_2 \Rightarrow C' \rrbracket$ 
    br %merge
merge:
```

# Connecting this to Code

- Instruction streams:
  - Must include labels, terminators, and “hoisted” global constants
- Must post-process the stream into a control-flow-graph
- See frontend.ml from HW4



# OPTIMIZING CONTROL

# Standard Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
    z = 3;
else
    z = 4;
return z;
```



```
%tmp1 = icmp Eq [[y]], 0      ; !y
%tmp2 = and [[x]] [[tmp1]]
%tmp3 = icmp Eq [[w]], 0
%tmp4 = or %tmp2, %tmp3
%tmp5 = icmp Eq %tmp4, 0
br %tmp4, label %else, label %then

then:
    store [[z]], 3
    br %merge

else:
    store [[z]], 4
    br %merge

merge:
    %tmp5 = load [[z]]
    ret %tmp5
```

# Observation

- Usually, we want the translation  $\llbracket e \rrbracket$  to produce a value
  - $\llbracket C \vdash e : t \rrbracket = (\text{ty}, \text{operand}, \text{stream})$
  - e.g.  $\llbracket C \vdash e_1 + e_2 : \text{int} \rrbracket = (\text{i64}, \%tmp, [\%tmp = \text{add } \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket])$
- But when the expression we're compiling appears in a test, the program jumps to one label or another after the comparison but otherwise never uses the value.
- In many cases, we can avoid “materializing” the value (i.e. storing it in a temporary) and thus produce better code.
  - This idea also lets us implement different functionality too:  
e.g. short-circuiting boolean expressions



# Idea: Use a different translation for tests

Usual Expression translation:

$\llbracket C \vdash e : t \rrbracket = (\text{ty}, \text{operand}, \text{stream})$

Conditional branch translation of booleans,  
without materializing the value:

$\llbracket C \vdash e : \text{bool}@ \rrbracket \text{ ltrue lfalse} = \text{stream}$

$\llbracket C, \text{rt} \vdash \text{if } (e) \text{ then } s_1 \text{ else } s_2 \Rightarrow C' \rrbracket = \llbracket C' \rrbracket,$

```
    insns3
  then:
     $\llbracket s_1 \rrbracket$ 
    br %merge
  else:
     $\llbracket s_2 \rrbracket$ 
    br %merge
  merge:
```

Notes:

- takes two extra arguments: a “true” branch label and a “false” branch label.
- Doesn’t “return a value”
- Aside: this is a form of continuation-passing translation...

where

$\llbracket C, \text{rt} \vdash s_1 \Rightarrow C' \rrbracket = \llbracket C' \rrbracket, \text{ insns}_1$

$\llbracket C, \text{rt} \vdash s_2 \Rightarrow C'' \rrbracket = \llbracket C'' \rrbracket, \text{ insns}_2$

$\llbracket C \vdash e : \text{bool}@ \rrbracket \text{ then else} = \text{insns}_3$

# Short Circuit Compilation: Expressions

- $\llbracket C \vdash e : \text{bool}@ \rrbracket \text{ ltrue lfalse} = \text{insns}$

---

$$\llbracket C \vdash \text{false} : \text{bool}@ \rrbracket \text{ ltrue lfalse} = [\text{br } \%1\text{false}] \quad \text{FALSE}$$

---

$$\llbracket C \vdash \text{true} : \text{bool}@ \rrbracket \text{ ltrue lfalse} = [\text{br } \%1\text{true}] \quad \text{TRUE}$$

---

$$\llbracket C \vdash e : \text{bool}@ \rrbracket \text{ lfalse ltrue} = \text{insns}$$

---

$$\llbracket C \vdash !e : \text{bool}@ \rrbracket \text{ ltrue lfalse} = \text{insns} \quad \text{NOT}$$

# Short Circuit Evaluation

Idea: build the logic into the translation

$$\llbracket C \vdash e1 : \text{bool}@ \rrbracket \text{ ltrue right} = \text{insns}_1 \quad \llbracket C \vdash e2 : \text{bool}@ \rrbracket \text{ ltrue lfalse} = \text{insns}_2$$
$$\llbracket C \vdash e1 \mid e2 : \text{bool}@ \rrbracket \text{ ltrue lfalse} =$$

```
insns1
right:
  insn2
```

$$\llbracket C \vdash e1 : \text{bool}@ \rrbracket \text{ right lfalse} = \text{insns}_1 \quad \llbracket C \vdash e2 : \text{bool}@ \rrbracket \text{ ltrue lfalse} = \text{insns}_2$$
$$\llbracket C \vdash e1 \& e2 : \text{bool}@ \rrbracket \text{ ltrue lfalse} =$$

```
insns1
right:
  insn2
```

where `right` is a fresh label

# Short-Circuit Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
    z = 3;
else
    z = 4;
return z;
```



```
%tmp1 = icmp Eq [[x]], 0
br %tmp1, label %right2, label %right1

right1:
    %tmp2 = icmp Eq [[y]], 0
    br %tmp2, label %then, label %right2

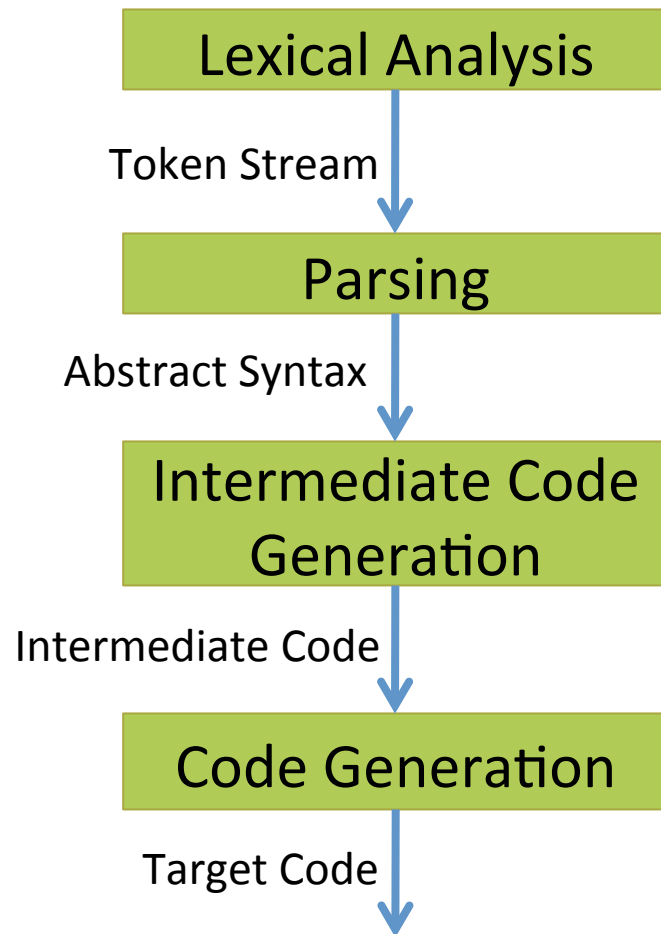
right2:
    %tmp3 = icmp Eq [[w]], 0
    br %tmp3, label %then, label %else

then:
    store [[z]], 3
    br %merge

else:
    store [[z]], 4
    br %merge

merge:
    %tmp5 = load [[z]]
    ret %tmp5
```

# The Story So Far



- As of HW4:
  - See how to compile a C-like language to x86 assembly by way of the LLVM IR
- Main idea 1:
  - Translation by way of a series of languages, each with well-defined semantics
- Main idea 2:
  - Structure of the semantics (e.g. scoping and/or type-checking rules) guides the structure of the translation

# What's next?

- Source language features:

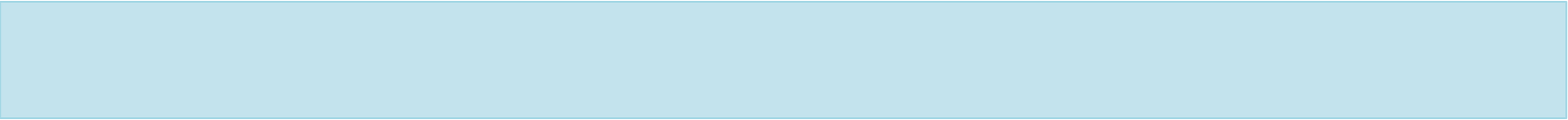
- First-class functions
- Objects & Classes
- Polymorphism
- Modules

⇒ How do we define their semantics? How do we compile them?

- Performance / Optimization:

- How can we improve the quality of the generated code?
- What information do we need to do the optimization?

⇒ Static analyses



Untyped lambda calculus  
Substitution  
Evaluation

# FIRST-CLASS FUNCTIONS

# “Functional” languages

- Languages like ML, Haskell, Scheme, Python, C#, Java 8, Swift
- Functions can be passed as arguments (e.g. map or fold)
- Functions can be returned as values (e.g. compose)
- Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1
```

```
let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

- How do we implement such functions?



# Free Variables and Scoping

```
let add = fun x -> fun y -> x + y
let inc = add 1
```

- The result of `add 1` is a function
- After calling `add`, we can't throw away its argument (or its local variables) because those are needed in the function returned by `add`.
- We say that the variable `x` is *free* in `fun y -> x + y`
  - Free variables are defined in an outer scope
- We say that the variable `y` is *bound* by “`fun y`” and its scope is the body “`x + y`” in the expression `fun y -> x + y`
- A term with no free variables is called *closed*.
- A term with one or more free variables is called *open*.

# (Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language.
  - Note: we're writing `(fun x -> e)` lambda-calculus notation:  $\lambda x. e$
- It has variables, functions, and function application.
  - That's it!
  - It's Turing Complete.
  - It's the foundation for a *lot* of research in programming languages.
  - Basis for “functional” languages like Scheme, ML, Haskell, etc.

Abstract syntax in OCaml:

```
type exp =  
  | Var of var          (* variables *)  
  | Fun of var * exp    (* functions: fun x -> e *)  
  | App of exp * exp    (* function application *)
```

Concrete syntax:

```
exp ::=  
  | x          variables  
  | fun x -> exp functions  
  | exp1 exp2 function application  
  | ( exp )    parentheses
```

# Values and Substitution

- The only values of the lambda calculus are (closed) functions:

```
val ::=  
    | fun x -> exp    functions are values
```

- To *substitute* a (closed) value  $v$  for some variable  $x$  in an expression  $e$ 
  - Replace all *free occurrences* of  $x$  in  $e$  by  $v$ .
  - In OCaml: written `subst v x e`
  - In Math: written  $e\{v/x\}$

- Function application is interpreted by *substitution*:

```
(fun x -> fun y -> x + y) 1  
= subst 1 x (fun y -> x + y)  
= (fun y -> 1 + y)
```

# Lambda Calculus Operational Semantics

- Substitution function (in Math):

$x\{v/x\}$	$= v$	<i>(replace the free <math>x</math> by <math>v</math>)</i>
$y\{v/x\}$	$= y$	<i>(assuming <math>y \neq x</math>)</i>
$(\text{fun } x \rightarrow \text{exp})\{v/x\}$	$= (\text{fun } x \rightarrow \text{exp})$	<i>(<math>x</math> is bound in <math>\text{exp}</math>)</i>
$(\text{fun } y \rightarrow \text{exp})\{v/x\}$	$= (\text{fun } y \rightarrow \text{exp}\{v/x\})$	<i>(assuming <math>y \neq x</math>)</i>
$(e_1 \ e_2)\{v/x\}$	$= (e_1\{v/x\} \ e_2\{v/x\})$	<i>(substitute everywhere)</i>

- Examples:

$$x \ y \ \{(\text{fun } z \rightarrow z)/y\} \Rightarrow x \ (\text{fun } z \rightarrow z)$$

$$(\text{fun } x \rightarrow x \ y)\{(\text{fun } z \rightarrow z) / y\} \Rightarrow (\text{fun } x \rightarrow x \ (\text{fun } z \rightarrow z))$$

$$(\text{fun } x \rightarrow x)\{(\text{fun } z \rightarrow z) / x\} \Rightarrow (\text{fun } x \rightarrow x) \quad // \ x \text{ is not free!}$$

# Free Variable Calculation

- An OCaml function to calculate the set of free variables in a lambda expression:

```
let rec free_vars (e:exp) : VarSet.t =  
  begin match e with  
    | Var x          -> VarSet.singleton x  
    | Fun(x, body)   -> VarSet.remove x (free_vars body)  
    | App(e1, e2)    -> VarSet.union (free_vars e1) (free_vars e2)  
  end
```

- A lambda expression  $e$  is *closed* if `free_vars e` returns `VarSet.empty`
- In mathematical notation:

$$\begin{aligned} \text{fv}(x) &= \{x\} \\ \text{fv}(\text{fun } x \rightarrow \text{exp}) &= \text{fv}(\text{exp}) \setminus \{x\} \quad (\text{'x' is a bound in exp}) \\ \text{fv}(\text{exp}_1 \text{ exp}_2) &= \text{fv}(\text{exp}_1) \cup \text{fv}(\text{exp}_2) \end{aligned}$$

# Operational Semantics

- Specified using just two inference rules with judgments of the form  $\text{exp} \Downarrow \text{val}$ 
  - Read this notation as “program  $\text{exp}$  evaluates to value  $\text{val}$ ”
  - This is *call-by-value* semantics: function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

“Values evaluate to themselves”

$$\frac{\text{exp}_1 \Downarrow (\text{fun } x \rightarrow \text{exp}_3) \quad \text{exp}_2 \Downarrow v \quad \text{exp}_3\{v/x\} \Downarrow w}{\text{exp}_1 \text{ exp}_2 \Downarrow w}$$

“To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. ”

# Variable Capture

- Note that if we try to naively "substitute" an open term, a bound variable might capture the free variables:

$$\begin{aligned} & (\text{fun } x \rightarrow (x \ y)) \ \{(\text{fun } z \rightarrow x) / y\} \\ = & \text{fun } x \rightarrow (x \ (\text{fun } z \rightarrow x)) \end{aligned}$$

Note:  $x$  is free in  $(\text{fun } x \rightarrow x)$   
free  $x$  is *captured!!*

- Usually *not* the desired behavior
  - This property is sometimes called "dynamic scoping"  
The meaning of " $x$ " is determined by where it is bound dynamically, not where it is bound statically.
  - Some languages (e.g. emacs lisp) are implemented with this as a "feature"
  - But, leads to hard to debug scoping issues

# Alpha Equivalence

- Note that the names of bound variables don't matter.
  - i.e. it doesn't matter which variable names you use, as long as you use them consistently

(fun **x** -> y **x**) is the "same" as (fun **z** -> y **z**)

the choice of "x" or "z" is arbitrary, as long as we consistently rename them

- Two terms that differ only by consistent renaming of bound variables are called *alpha equivalent*
- The names of free variables do matter:

(fun x -> **y** x) is *not* the "same" as (fun x -> **z** x)

Intuitively: y and z can refer to different things from some outer scope



# Fixing Substitution

- Consider the substitution operation:  
 $\{e_2/x\} e_1$
- To avoid capture, we define substitution to pick an alpha equivalent version of  $e_1$  such that the bound names of  $e_1$  don't mention the free names of  $e_2$ .
  - Then do the "naïve" substitution.

For example:  $(\text{fun } x \rightarrow (x \ y)) \{(\text{fun } z \rightarrow x) / y\}$   
 $= (\text{fun } x' \rightarrow (x' (\text{fun } z \rightarrow x)))$       *rename x to x'*